FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

EARL
A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs

Felix Wolf, Bernd Mohr

FZJ-ZAM-IB-9803

April 1998

(letzte Änderung: 01.04.98)

EARL

– A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs –

Felix Wolf **Bernd Mohr**
Forschungszentrum Jülich GmbH
ZAM, 52425 Jülich, Germany
f.wolf@fz-juelich.de **b.mohr@fz-juelich.de**

Abstract. This paper describes a new meta-tool named EARL which consists of a new high-level trace analysis language and its interpreter which allows to easily construct new trace analysis tools. Because of its programmability and flexibility, EARL can be used for a wide range of event trace analysis tasks. It is especially well-suited for automatic and for application or domain specific trace analysis and program validation. We describe the abstract view on an event trace the EARL interpreter provides to the user, and give an overview about the EARL language. Finally, a set of EARL script examples are used to demonstrate the features of EARL.

1 Motivation

Using event tracing to analyze the behavior of parallel and distributed applications is a well accepted technique. In addition, there are a multitude of powerful, graphical event trace analysis tools (e.g., AIMS[9], Paradyn[8], Pablo[6], SIMPLE[5], VAMPIR[3], Upshot[4], and many more). However, all of them have one or more of the following shortcomings:

1. The biggest problem (especially with graphical tools) is that event traces generated on today's large and fast machines are getting very big. Either the tools show the recorded behavior by displaying an animation or they read in the whole trace at once, display it, and then allow the user to zoom in and out. If analyzing very large traces, a user looking for problems/bottlenecks would either have to watch or zoom in and out for a long time. In the second case, it is possible that the trace is too large to be read in in total. In either case, the user never knows whether he missed something because he didn't look carefully enough or zoomed in at the wrong places. Clearly, a more "automatic" way of analyzing large traces is needed.
2. Although the tools provide a large number of graphical views the right one needed for the application might not be available. Typically, many of these tools cannot be used if the analysis has to be carried out in a domain or application specific way not covered by the graphical displays provided with the tools. Also, many cannot handle user-defined events in a useful way. A more flexible and easily programmable tool is needed.

   Such a tool would also allow a tool expert to explore new ideas for trace analysis tools or to quickly implement custom-made tools for ordinary users if needed.
3. Traditional performance analysis tools have very little or no support for conducting experiments (i.e., repeated measurements with varying processor numbers or input data sets). Often, they cannot analyze more than one event trace at the same time (e.g., to support trace comparisons).

We therefore designed and implemented a new meta-tool named EARL (**E**vent **A**nalysis and **R**ecognition **L**anguage). EARL is actually a new high-level trace analysis language which allows to easily construct new trace analysis tools by writing scripts in the EARL language. These are then executed by the EARL interpreter. Section 2 describes the EARL language and the implementation of the interpreter in more detail. Three longer EARL script examples in Section 3 show how easy it is to use EARL to implement new trace analysis tools. Section 4 discusses related work and Section 5 concludes the paper and describes the enhancements to EARL we hope to implement in the future.

2 The EARL Toolkit

In order to achieve the highest degree of flexibility and programmability for analyzing event traces, we designed and implemented a new high-level trace analysis language. Although EARL is designed to be a generic event trace analysis tool, the current prototype (which is described in this paper) concentrates on the analysis of event traces generated from message passing programs. This is not really a restriction as most uses of event tracing are in the field of parallel programming on distributed memory machines (which almost all use a one or two sided message passing scheme for communication). In addition, analysis of message passing traces is well understood and therefore allowed us to provide high-level, well known abstractions as the programming interface to an EARL user.

2.1 Abstract View on an Event Trace

Much of the power of EARL comes through its very high-level abstraction of an event trace which allows a programmer to concentrate on the trace analysis and let EARL take care of the different trace formats and their encoding of functions and event types, of input handling and buffering, and of keeping track of message queues and call stacks.

An EARL programmer can view an event trace as a sequence of *events*. The events are sorted according to their timestamp and numbered starting from 1. There are different *event types*. EARL defines four predefined event types: entering (named `enter`) and leaving (`exit`) a region, and sending (`send`) and receiving (`recv`) a message. There may be more event types defined depending on the underlying trace format. A *region* is a named section of the traced program (e.g., it could be a loop or basic block, but mostly it is a function or subroutine). If supported by the trace format, regions may be organized in *groups* (e.g., user or system functions).

An event type is represented by a n-tuple of *attributes*. An event (instance) is defined by a corresponding n-tuple of values assigned to these attributes. The number of attributes depends on the type of the event. However, all event types have the following attributes in common:

`num:` The number of the event.

`node:` The location (cpu, pe, node) where the event happened. Nodes are numbered for `0` to `n-1` where `n` is the total number of locations used by the parallel program.

`time:` The timestamp of the event as a floating point value in seconds.

`type:` The event type is explicitly given as a attribute value.

`enterptr:` The number of the `enter` event which determines the region in which the event happened. For `exit` events, this means that their `enterptr` refers to the matching `enter`.

The `enter` and `exit` event types have additional `region` and `group` attributes specifying the name of the region entered or left and its group, and `send` and `recv` have attributes describing the destination (`dest`), source (`src`), tag (`tag`), length (`len`), and communicator (`com`) of the message. In addition, the `recv` event type has a `sendptr` attribute pointing to the corresponding `send` event.

In addition to the basic event trace model, EARL provides the concepts of *regions* and *messages*. These are defined as pairs of matching events: `enter/exit` or `send/recv` respectively. In the EARL language, these concepts are supported in the form of the `enterptr` and `sendptr`. For each position in the event trace, EARL also defines a *region stack* per node and a *message queue* implemented as lists of enter and send events which define the regions entered and messages not yet received at that time. In addition to query theses structures directly, it is possible to navigate step-by-step through the region stack using the `enterptr` attribute or to trace back messages by following the `sendptr` attribute (see Figure 1) .

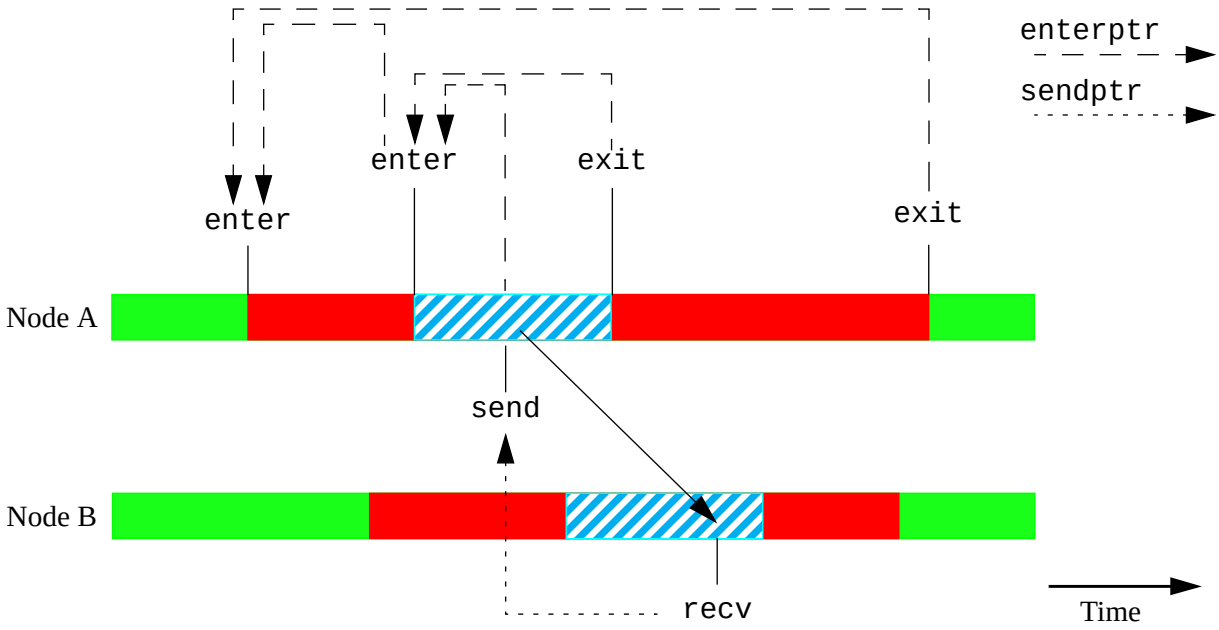

FIGURE 1. References provided by `sendptr` and `enterptr`

All these facilities together allow to easily process complex event patterns made out of regions and messages. This is demonstrated by the examples in Section 3.

2.2 Implementation Notes

The EARL interpreter reads and decodes the underlying trace format and maps it automatically to the EARL event types and attributes. This allows the programmers to write their trace analysis scripts independent from the format of the event trace and of the encoding of event types and function/region names. Currently, EARL supports the VAMPIR [3] and ALOG [4] trace formats.

Instead of re-inventing the wheel when implementing the EARL language, we started with the well known scripting language TCL [2] and extended it with commands for event trace and event record handling. The extensions are implemented in C++. The reasons for choosing TCL were:

- TCL was originally designed as extensible tool command language, and therefore was easy to extend.
- As a high-level scripting language, it allows rapid prototyping.
- TCL is very portable (it runs on UNIX, Macintosh, and Windows systems)
- There are already many useful extensions to TCL (e.g., the graphical toolkit TK, piecharts, bargraphs) which can also be used for the development of trace analysis tools.

- TCL comes with built-in interprocess communication (the TCL built-in `send` command and sockets) which makes it easy to integrate it with other trace analysis tools or programming environments (like TAU [10]).
- Like other Unix scripting languages, TCL allows to execute and control other processes very easily making it very suitable to implement trace analysis experimentation tools.

For efficient random access to events, EARL automatically buffers the most recently processed events in the *history buffer* and stores important trace state information (including the region stacks and the message queue) at fixed intervals in so-called *bookmarks*.

2.3 List of EARL Commands

This section gives an overview of the new commands we added to TCL in order to allow high-level, portable, and efficient event trace analysis. The EARL extensions follow the object-oriented style which is also used in TK: the command to open a event trace returns a *trace object handle* which is automatically registered as a new TCL command. The other EARL functions are implemented as methods of the trace object. EARL supports the following functionality:

Trace handling: The `earl open` command takes the filename of the event trace as an argument and returns a handle to it. It has optional switches to pass the trace format (`-format`), the size of the history buffer (`-hist`), and the distance between bookmarks (`-mark`). The `close` method closes the event trace and releases all related resources.

Event access: EARL provides two methods for reading events: the `get` method returns an event as a list {$attr_1$ $value_1$ $attr_2$ $value_2$ ... $attr_n$ $value_n$}, while the `set` method fills an specified associative array `arr` in a way that `arr(`$attr_i$`) ==` $value_i$. Both take the number of the event to process as argument. In addition, they set the *current event pointer* to the processed event unless the optional switch `-fetchonly` is used. Both methods come in three flavors: the user can pass the number of the event to process, or move through the trace sequentially forward or backward (relative to the current event pointer) by using the additional methods `setnext` and `getnext` or `setprev` and `getprev`.

State access: EARL automatically keeps track of the state of the region stacks and the message queue for the current event. The `stack` method returns the stack of a specified node as a list of either the region names (`-sym` switch) or the event number of the corresponding `enter` events (default). The `queue` method returns the message queue as a list of event numbers which point to the corresponding `send` events.

General information access: The `info` method gives access to general information about the event trace. It allows to get a list of all defined event types (`eventtypes`), a list of all defined attributes for a specified type (`attributes`), the filename (`filename`) and format (`format`) of the event trace, the number of nodes used in the parallel application (`nodecount`), and a list of all defined regions (`regions`) and groups (`groups`).

Statistics: Event trace analysis often involves keeping statistics of a large number of values like the durations of a region or the transfer rates of messages. EARL supports this by providing *statistic objects*. The command `earl stat` creates a new statistic object and returns a handle to it. The method `addval` adds a new value to the data set. At any point, the user can ask for the number of values in the data set (method `count`), the minimum (`min`), maximum (`max`), mean (`mean`), median (`med`), sum (`sum`), variance (`var`), and the 25% and 75% quantiles (`q25, q75`). The quantiles (`med, q25, q75`) are actually estimates computed with the P^2 algorithm [11] which makes it unnessary to store the complete dataset. Finally, there are methods to `reset` or `delete` statistic objects.

3 EARL Script Examples

This section describes two EARL script examples. Each of them is generic in the sense that it can be used with any message passing trace supported by EARL. Although simple (all are around 20 lines of code) they perform quite complex calculations. The simplicity comes from the abstractions defined in the EARL event trace model and the high-level nature of the TCL scripting language.

3.1 Example 1: Region Statistics

The first problem is the standard task of computing the time which is spent in each region of the program including and excluding the time spent in contained regions[1]:

```
 1:   #!/usr/local/bin/earl
 2:   set t [earl open [lindex $argv 0]]
 3:   set n [$t info nodecount]
 4:   for {set i 0} {$i<$n} {incr i} {
 5:     foreach r [$t info regions] {
 6:       set incl($i,$r) 0
 7:       set excl($i,$r) 0
 8:     }
 9:   }
10:   while {[$t setnext curr] != -1} {
11:     if {$curr(type) == "exit"} {
12:       $t set enter $curr(enterptr) -fetchonly
13:       set diff [expr $curr(time) - $enter(time)]
14:       set index "$curr(node),$curr(region)"
15:       set incl($index) [expr $incl($index) + $diff]
16:       set excl($index) [expr $excl($index) + $diff]
17:       set parent [lindex [$t stack $curr(node) -sym] 0]
18:       if {$parent != ""} {
19:         set excl($curr(node),$parent) \
20:                         [expr $excl($curr(node),$parent) - $diff]
21:       }
22:     }
23:   }
24:   $t close
```

Line 1 is a special comment which tells a Unix system which command to use to execute the following script file. Line 2 opens a trace file which is given the script as first command line parameter and stores the handle in variable `t`. Next, we get the number of nodes used (line 3) and a list of all regions defined (line 5). With this information, we use a double loop (lines 4 to 9) to initialize the two arrays `incl` and `excl`, where we store the inclusive and exclusive time spent in each region per node.

The `while` in line 10 steps sequentially through the event trace setting the array `curr` to the next event. If we find an `exit` event (line 11), we fill the array `enter` with the corresponding `enter` event of the region we are about to leave (line 12). In line 13 we calculate the time spent in this region. In lines 15 and 16, we add this time to the corresponding values in the arrays `incl` and `excl`. Here we use the auxiliary variable `index` which we computed in line 14. Lines 17 to 21 substract the execution time of the current region from the corresponding exclusive execution time of the enclosing region (if there is one). In order to

1. The line numbers are not part of the source code.

find it, we use the `stack` method to get the region stack of the current event and node (line 17); but we only use the first entry which is the parent region and store it in the variable `parent`. For this we use the TCL command "`lindex ... 0`" which extracts the entry at index 0. Finally, we close the trace in line 24. At this point, the arrays `incl` and `excl` contain the desired information which now can be printed or displayed using bargraphs or piecharts.

3.2 Example 2: Compute Wasted Time of MPI_Recv

The second example demonstrates the capabilities of EARL for solving non-standard problems, especially recognizing complex events patterns. Consider the following: For a set of event traces from a parallel MPI program, determine the time which is wasted when a `MPI_Recv` is posted before the corresponding `MPI_Send` was executed (see Figure 2). .

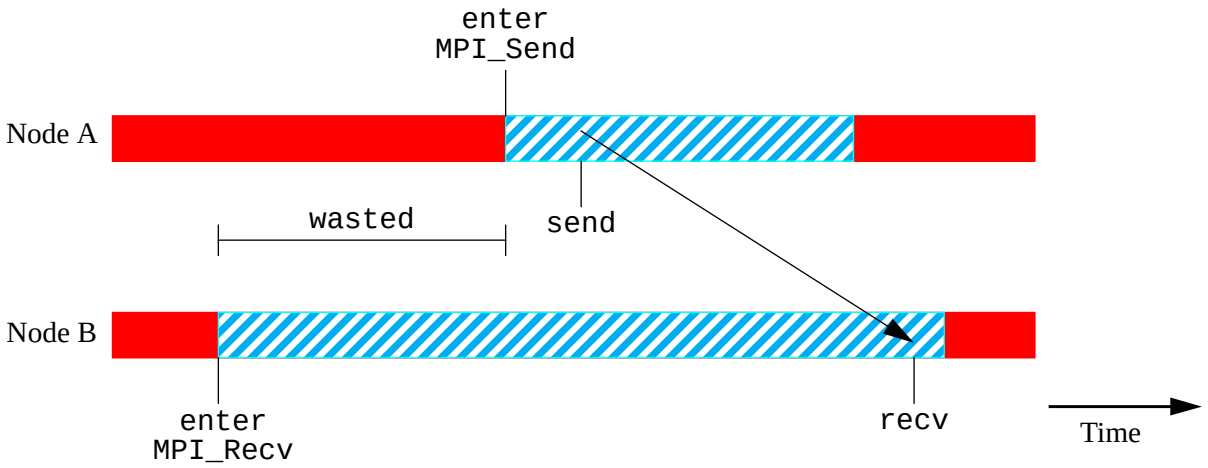

FIGURE 2. Wasted Time in Message Passing Programs

Here is the complete EARL script code:

```
 1:   #!/usr/local/bin/earl
 2:   foreach arg $argv {
 3:     set t [earl open $arg]
 4:     set sum_wasted 0
 5:     while {[$t setnext curr] != -1} {
 6:       if {$curr(type) == "recv"} {
 7:         $t set recv_start $curr(enterptr) -fetchonly
 8:         if {$recv_start(region) != "MPI_Recv"} continue
 9:         $t set send $curr(sendptr) -fetchonly
10:         $t set send_start $send(enterptr) -fetchonly
11:         if {$send_start(region) != "MPI_Send"} continue
12:         set wasted [expr $send_start(time) - $recv_start(time)]
13:         if {$wasted>0} {
14:           set sum_wasted [expr $sum_wasted + $wasted]
15:         }
16:       }
17:     }
18:     puts "[$t info filename]:  $sum_wasted seconds wasted."
19:     $t close
20:   }
```

Line 2 loops through a set of trace files specified as command line arguments. Line 3 opens the trace file which is specified by the current command line parameter `arg` and stores the handle in variable `t`. The `while` in line 5 steps sequentially through the event trace setting the array `curr` to the next event. If we find a `recv` event (line 6), we fill the array `recv_start` with the `enter` event of the enclosing region (line 7). If the enclosing region is not `MPI_Recv` (the message could have been sent from another routine, e.g., `MPI_Broadcast`), we skip the rest of the loop and continue the search (line 8). Next, we set array `send` to the corresponding `send` event (line 9), and again check whether it originated from a `MPI_Send` (lines 10 and 11). We compute the difference between the begin of `MPI_send` and `MPI_Recv` (line 12) and add it to the variable sum_`wasted` if `MPI_Recv` executed before `MPI_Send` (line 14). Finally, we print the result (line 18) and close the trace (line 19).

3.3 Example 3: Passing Messages Out of Order

The final example demonstrates how EARL can be used to find programming errors in message passing programs. The example is taken from the Grindstone test suite for parallel performance tools [13] and highlights the problem of passing messages "out-of-order". This problem could arise if one process is expecting messages in a certain order, but another process is sending messages which are not in the expected order. In Figure 3, an extreme example is shown: in the first part of the program, Node 1 is processing incoming messages in the opposite order they were sent from Node 0. Processing them in the order they were sent would not only speed-up the program but also requires much less buffer space for storing unprocessed messages.

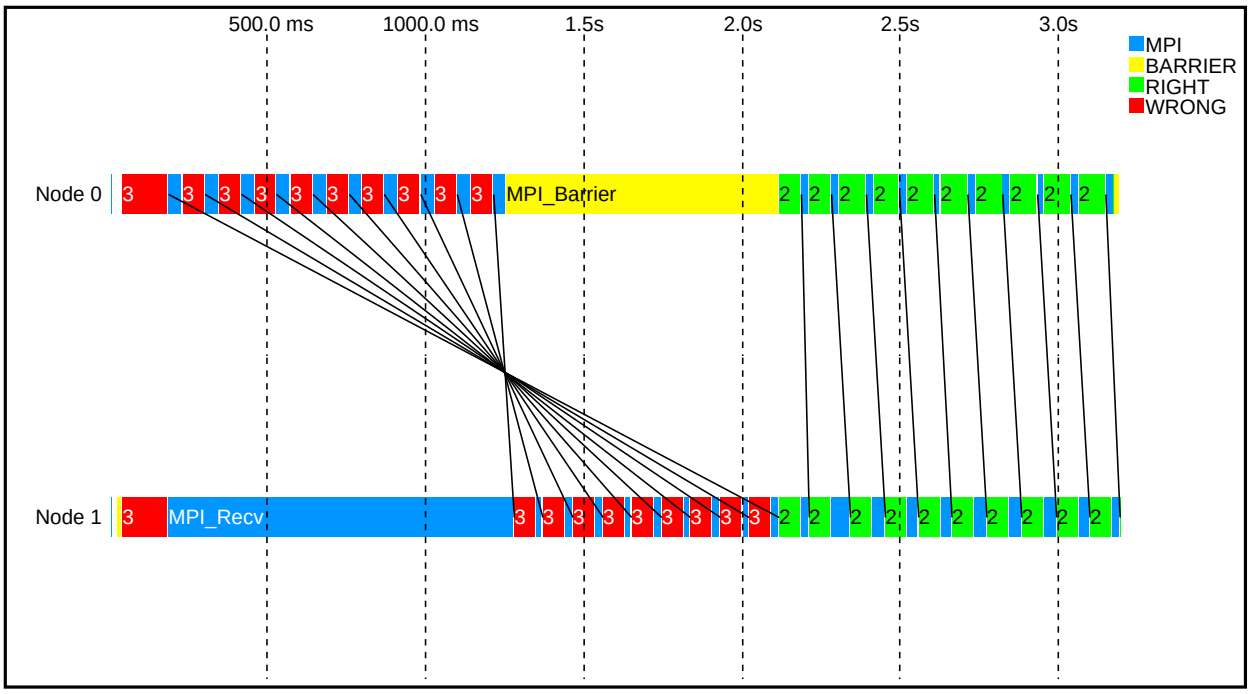

FIGURE 3. Passing Messages Out of Order

The EARL code for this example is trivial:

```
1:   #!/usr/local/bin/earl
2:   set t [earl open [lindex $argv 0]]
3:   while {[$t setnext curr] != -1} {
4:     if {$curr(type) == "recv"} {
```

```
 5:        foreach send [$t queue $curr(node) $curr(src)] {
 6:          if {$send < $curr(sendptr)} {
 7:            puts "Received message in wrong order:"
 8:            puts "  on node $curr(node) at $curr(time)"
 9:            puts "  call stack: [$t stack $curr(node) -sym]"
10:            break
11:          }
12:        }
13:      }
14:    }
15:    $t close
```

Like in the first example, we open the trace file specified as first command line parameter (line 2) and sequentially loop through the events of the trace (line 3). If we find a `recv` event (line 4), we check all messages still in the message queue sent to the current node from the same source node as the current message (line 5), whether the corresponding `send` event happened before the `send` event of the current message (line 6). As we only have to determine the order of these events in time and EARL provides references to other events in the form of event numbers, the necessary comparison can be done by comparing the references. If we find such an outstanding message, an error message is printed (line 7 to 9).

4 Related Work

EARL is certainly not the first programmable tool for event trace analysis:

- EDL [1] was one of the first trace analysis tool which was programmable. EDL allows to define custom hierarchies of events based on regular event expressions. The expressions were translated into an automaton which tried to locate the defined events in an event trace. While working well for sequential programs, parallel programs required the use of a special interleaving operator. The use of this operator results in huge automatons so the tool can only be used with small and short parallel programs or simple parallel patterns only.
- SIMPLE [5] is an environment for event trace analysis. It includes a large set of tools for specific tasks, each of which defines its own command language for adapting it to specific trace formats or application areas. Although very powerful, this makes the usage of SIMPLE very complex. Also, it is not possible to combine the tasks of the different tools, so that in the worst case, each task requires the processing of the whole event trace. EARL on the other side, allows to combine different scripts, so that an event trace must be read only once.
- Pablo [6] is a very powerful, programmable, graphical event trace analysis tool. Pablo can be programmed by arranging predefined modules for trace input, event record processing, and visualization in a configuration window and connecting them. The modules are highly configurable. It is simple to use as long as the desired analysis matches the intended use of the predefined modules otherwise the graphical programming can be cumbersome or difficult. It cannot easily be extended because the implementation of new (user-defined) modules is quite complex.

Paradyn [8] is not a generic event trace analysis tool but rather a tool for performance analysis and optimization of parallel programs. It automatically tries to locate performance bottlenecks. Trace overhead and size is kept low by dynamic and selective instrumentation. It is also one of the few tools which support experiment management [12].

5 Conclusion and Future Work

We just completed our first prototype of EARL. Early experiments show that although simple in design, EARL is a powerful and easy to use meta-tool for experts[2] to implement generic or custom-made program or application domain specific event trace analysis tools. Because of its programmability and flexibility, EARL can be used for a wide range of event trace analysis tasks:

- calculation of performance indices and trace statistics of all kinds
- finding all locations of possible bottlenecks (which then can be analyzed with traditional graphical trace analysis tools if necessary)
- performance visualization and animation (as far as TK or other TCL graphics extensions are suitable for this task)
- experiment management where within an experiment the instrumentation of the parallel programs and generation of traces is based on results calculated from earlier runs. This allows to implement automatic program optimization tools.
- application or domain specific versions of these tasks

In the next months, we want to implement a library of useful generic EARL scripts and subroutines which then can be used by programmers to analyze their parallel applications. We also hope to implement additional decoder modules for other trace formats (e.g., PICL [7] or SDDF [6]) and to add more direct support for traces generated by programs in other programming paradigms than message passing.

In addition, we recently started a new project to design and implement an environment for the automatic detection of standard bottlenecks in parallel or distributed applications called KOJAK (**K**it for **O**bjective **J**udgement and **A**utomatic **K**nowledge-based detection of bottlenecks). In this project, we plan to explore different ways of representing and locating bottlenecks. Here, we plan to use EARL in order to easily implement and evaluate the different methods.

6 References

[1] P. Bates, *Debugging Programs in a Distributed System Environment*, Ph.D. Thesis, University of Massachusetts, February 1986.

[2] J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[3] A. Arnold, U. Detert, and W.E. Nagel, Performance Optimization of Parallel Programs: Tracing, Zooming, Understanding, in: R. Winget and K. Winget, editors, *Proc. Cray User Group Meeting*, Spring 1995, pages 252-258, Denver, CO, March 1995.

[4] V. Herrarte and E. Lusk, *Studying Parallel Program Behavior with Upshot*, Technical Report ANL-91/15, Mathematics and Computer Science Division, Argonne National Laboratory, August 1991.

[5] B. Mohr, Standardization of Event Traces Considered Harmful or Is an Implementation of Object-Independent Event Trace Monitoring and Analysis Systems Possible? in: J.J. Dongarra and B. Tourancheau, editors, *Proc. CNRS-NSF Workshop on Environments and Tools For Parallel Scientific Computing*, volume 6 of *Advances in Parallel Computing*, pages 103-124, Elsevier, September 1992.

2. especially if they know TCL :-)

[6] Reed, D.A. and Olson, R.D. and Aydt, R.A. and Madhyasta, T.M. and Birkett, T. and Jensen, D.W. and Nazief, A.A. and Totty, B.K., Scalable Performance Environments for Parallel Systems, in: *Proc. 6th Distributed Memory Computing Conference*, pages 562-569, IEEE Computer Society Press, 1991.

[7] G.A. Geist, M.T. Heath, B.W. Peyton, and P.H. Worley, *PICL: A Portable Instrumented Communication Library*, Technical Report ORNL/TM-11130, Oak Ridge National Laboratory, Tennessee, July 1990.

[8] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K. Kunchithapadam, K.L. Karavanic, and T. Newhall, *The Paradyn Parallel Performance Measurement Tools*, IEEE Computer 28(11), November 1995.

[9] J. C. Yan, S. R. Sarukkai, and P. Mehra, *Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit*, Software Practice & Experience, Vol. 25, No. 4, pages 429-461, April 1995.

[10] B. Mohr, A. Malony, J. Cuny, TAU, in: G. V. Wilson, P. Lu, editors, *Parallel Programming Using C++*, pages 589-628, MIT Press, 1996.

[11] R. Jain, I. Chlamtac, *The P2 Algorithm for Dynamic Calculation of Quantiles and Histograms Without Storing Observations*, in: Communcations of the ACM, Vol. 28, No. 10, Oct 1985.

[12] K.L. Karavanic, B.P. Miller, Experiment Management Support for Performance Tuning, in: Proc. Supercomputing'97, San Jose, Nov 1997.

[13] J.K. Hollingsworth, M. Steele, *Grindstone: A Test Suite for Parallel Performance Tools*, Computer Science Technical Report CS-TR-3703, University of Maryland, Oct. 1996.